**Week4\_Create authentication service that returns JWT**

### Part 1: Environmental Setup - Adding Security Dependencies

Objective: To add the necessary libraries for Spring Security and JWT generation to the project's pom.xml.

#### Modifications to pom.xml

The following dependencies were added to enable security and token functionality:

<!-- 1. Spring Boot Starter for Security --><!-- This enables web security, protects all endpoints by default, and provides authentication filters. --><dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-security</artifactId></dependency>

<!-- 2. Java JWT (JJWT) Library --><!-- A standard library for creating, signing, and parsing JSON Web Tokens. --><dependency>

<groupId>io.jsonwebtoken</groupId>

<artifactId>jjwt-api</artifactId>

<version>0.11.5</version></dependency><dependency>

<groupId>io.jsonwebtoken</groupId>

<artifactId>jjwt-impl</artifactId>

<version>0.11.5</version>

<scope>runtime</scope></dependency><dependency>

<groupId>io.jsonwebtoken</groupId>

<artifactId>jjwt-jackson</artifactId>

<version>0.11.5</version>

<scope>runtime</scope></dependency>

### Part 2: Implementation Steps

#### Step 1: Core Security Configuration

Objective: To create a central security configuration to define users and security rules for API endpoints.

* New Class: SecurityConfig.java
* Location: com.cognizant.spring\_learn.config package

package com.cognizant.spring\_learn.config;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.security.config.annotation.web.builders.HttpSecurity;

Import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;

import org.springframework.security.core.userdetails.User;

import org.springframework.security.core.userdetails.UserDetails;

Import org.springframework.security.provisioning.InMemoryUserDetailsManager;

import org.springframework.security.web.SecurityFilterChain;

@Configuration@EnableWebSecurity

public class SecurityConfig {

// Defines an in-memory user store for demonstration.

@Bean

public InMemoryUserDetailsManager userDetailsService() {

UserDetails user = User.withUsername("user")

.password("{noop}pwd") // {noop} = No password encoding

.roles("USER")

.build();

return new InMemoryUserDetailsManager(user);

}

// Defines security rules for all HTTP requests.

@Bean

public SecurityFilterChain filterChain(HttpSecurity http) throwsException {

http

.csrf().disable() // Disable CSRF for stateless REST APIs.

.authorizeHttpRequests(auth -> auth

// Secure all endpoints. Every request must beauthenticated.

.anyRequest().authenticated()

)

// Enable HTTP Basic Authentication as the authenticationmethod.

.httpBasic();

return http.build();

}

}

#### Step 2: Authentication Controller and JWT Generation

Objective: To create the /authenticate endpoint that generates a JWT after Spring Security has successfully authenticated the user.

* New Class: AuthenticationController.java
* Location: com.cognizant.spring\_learn.controller package

package com.cognizant.spring\_learn.controller;

import java.util.Base64;import java.util.Date;

importjava.util.HashMap;import java.util.Map;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.RequestHeader;

import org.springframework.web.bind.annotation.RestController;

import io.jsonwebtoken.Jwts;

import io.jsonwebtoken.SignatureAlgorithm;

@RestController

public class AuthenticationController {

@GetMapping("/authenticate")

public Map<String, String> authenticate(@RequestHeader("Authorization") String authHeader) {

// Get user, generate token, and create response

String user = getUser(authHeader);

String token = generateToken(user);

Map<String, String> response = new HashMap<>();

response.put("token", token);

return response;

}

// Helper method to decode username from "Authorization: Basic ..." header

private String getUser(String authHeader) {

String base64Credentials = authHeader.substring("Basic".length()).trim();

byte[] credDecoded = Base64.getDecoder().decode(base64Credentials);

String credentials = new String(credDecoded);

// "user:pwd"

return credentials.split(":", 2)[0];

}

// Helper method to generate the JWT

private String generateToken(String user) {

String secretKey = "my-secret-key-that-is-long-enough"; // Store securely in real apps

return Jwts.builder()

.setSubject(user)

.setIssuedAt(new Date())

.setExpiration(new Date(System.currentTimeMillis() + 1000 \* 60 \* 20)) // Expires in 20 mins

.signWith(SignatureAlgorithm.HS256, secretKey)

.compact();

}

}

### Part 3: Testing and Verification

Objective: To test the new endpoint using the curl command line utility to verify both successful and failed authentication attempts.

#### Test Case 1: Successful Authentication (200 OK)

* Command: The -u user:pwd option tells curl to send an HTTP Basic Authentication header.

curl -u user:pwd

*http://localhost:8083/authenticate*

* Expected Response: A JSON object containing the signed JWT.

**Output**



#### Test Case 2: Failed Authentication (401 Unauthorized)

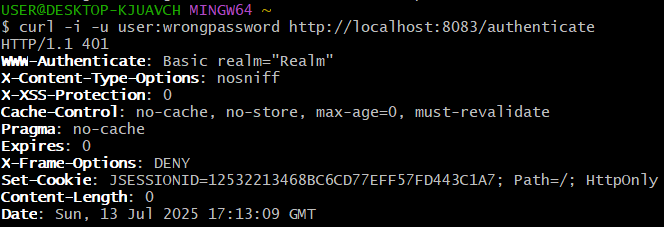
* Command: The -i flag is added to show the HTTP status code in the response. An incorrect password is used.

curl -i -u user:wrongpassword

*http://localhost:8083/authenticate*

* Expected Response: An HTTP 401 Unauthorized status, proving the security filter blocked the request before it reached the controller.

**Output**



#### Test Case 3: Postman (500 Unauthorized)

**Output**

